



DEPARTAMENTO  
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

# Trabajo Práctico

Programación concurrente

Sistemas Operativos  
Segundo Cuatrimestre de 2024

Integrante	LU	Correo electrónico
Dominguez, Maria Emilia	217/22	maemiliadominguez@gmail.com
Garcia Alurralde, Jorge	437/22	jalurralde@dc.uba.ar
Kerbs, Octavio Dario	64/22	octaviokerbs@gmail.com



Facultad de Ciencias Exactas y Naturales  
Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

<http://www.fcen.uba.ar>

# 1. Introducción

En el presente trabajo se busca dar solución a la gestión de la concurrencia para una página web que toma órdenes de compra. El principal desafío es capturar de cada orden qué productos y en qué cantidades se solicitan. La intención de diseño es obtener una estructura de datos que informe al administrador del sistema cuántos productos de cada tipo se vendieron. No es relevante para esta aplicación quiénes son los compradores, cuál es la dirección de envío ni el estado del pago.

La estructura de datos típica para almacenar tuplas (como sería: `<string tipo_de_producto, int cantidad>`), donde el primer elemento suele ser común a varias de las entradas, es el diccionario. Existen varias implementaciones para los diccionarios, como lo son los árboles binarios, ABL, heaps, etc. Para este trabajo se utilizará la implementación tabla hash.

La característica principal de las tablas hash es que las claves de almacenamiento son finitas. Sin embargo, las claves reales (los primeros elementos de las duplas) pueden ser más. Esto genera dos desafíos:

1. ¿Cómo se asigna cada clave a cada índice de la tabla hash?
2. Necesariamente varias claves van a coincidir con alguno de los índices, ¿cómo se discriminan las distintas claves de un mismo índice?

En primer lugar, se utiliza una función, llamada función hash, que vincula a cada clave con un índice de la tabla. Para este trabajo, la función hash reduce a cada producto a la letra minúscula con la que empieza su nombre. Los índices de la tabla van de 0 a 25, ordenándose las letras según el orden relativo en el código ASCII.

En segundo lugar, en caso de colisión, es decir, que dos o más claves coincidan con el mismo índice de la tabla, se define resolverlo a través de una lista enlazada. En cada nodo de la lista se almacena el par `<string tipo_de_producto, int cantidad>`.



Figura 1: Esquema general de tabla de hash

Las listas enlazadas implementadas en este trabajo reciben el nombre de `listaAtomica`. Estas estructuras están conformadas por nodos. Cada nodo almacena dos datos: el valor del nodo y el puntero al siguiente nodo. En caso de ser el último nodo de la lista, el valor del puntero siguiente debe ser `nullptr`. El valor almacenado por cada nodo es del tipo `hashMapPair`. Contiene el *string* con el tipo de producto y un *int* que representa la cantidad de productos.

Toda `listaAtomica` tiene por primer nodo al atributo privado `_cabeza`. Este nodo se caracteriza por ser el componente principal de la lista. Al inicializarse una lista, se asigna a su nodo `_cabeza` el valor `nullptr`. La particularidad que tiene `listaAtomica` sobre las listas enlazadas es que `_cabeza` es una estructura atómica. Esto quiere decir que solo un proceso va a poder operar de manera exclusiva con esta estructura, independientemente de la cantidad de procesos que lo soliciten.

Posición de memoria	Estructura de datos almacenada	Valor
*listaAtomica	listaAtomica<hashMapPair>	_cabeza._siguiente = 4000
...	...	...
4000	Nodo <hashMapPair>	nodo._valor = <"mandarina", 2> nodo._siguiente = 5000
...	...	...
5000	Nodo <hashMapPair>	nodo._valor = <"manzana", 6> nodo._siguiente = nullptr

Figura 2: Esquema general de listaAtomica

En las siguientes secciones de este trabajo se van a implementar distintas operaciones para las estructuras `HashMapConcurrente` y `listaAtomica`. Estas operaciones deben verificar ausencia de condiciones de carrera y deadlocks. Condiciones de carrera se refiere a una situación en la que el comportamiento del software depende del orden en que se ejecutan las operaciones de múltiples procesos que acceden a recursos compartidos. Un deadlock ocurre en cambio cuando dos o más procesos quedan bloqueados permanentemente esperando que el otro libere un recurso que necesitan para continuar.

## 2. Desarrollo

### 2.1. Ejercicio 1

El objetivo del primer ejercicio es implementar el método `insertar(T valor)`. Este método agrega un nuevo nodo al comienzo de la `listaAtomica`.

Para ejecutarlo correctamente, el método debe realizar las siguientes acciones:

1. Crear un nuevo nodo en memoria heap. Al nodo se le debe asignar `valor` como el `nodo._valor`.
2. Se le debe asignar a `nodo._siguiente` el valor de `lista._cabeza`.
3. Se debe actualizar el valor de `lista._cabeza` por la posición de memoria del nuevo nodo.

Ejecutados estos tres pasos, la inserción del nuevo nodo terminaría de forma correcta.

Posición de memoria	Estructura de datos almacenada	Valor
*listaAtomica	listaAtomica<hashMapPair>	_cabeza.siguiete = <del>4000</del> 6000
...	...	...
4000	Nodo <hashMapPair>	nodo.valor = <"mandarina", 2> nodo.siguiete = 5000
...	...	...
5000	Nodo <hashMapPair>	nodo.valor = <"manzana", 6> nodo.siguiete = nullptr
...	...	...
6000	Nodo<hashMapPair>	nodo.valor = <"mora", 1> nodo.siguiete = 4000

→ **Nodo Insertado**

Figura 3: Esquema para el método insertar

Recordando que estamos en un ambiente de concurrencia, podría ocurrir que haya más de un proceso insertando nodos al mismo tiempo. Si no se toman políticas de sincronización, varios nodos podrían definir que su valor de `nodo.siguiete` sean todos iguales. Esto generaría que sólo uno de los nodos insertados sea efectivamente agregado a la lista. Los demás no formarían parte de la cadena de recorrido.

### 2.1.1. Preguntas del TP

Se procede a responder las preguntas indicadas en la consigna:

- ¿Qué significa que la lista sea atómica?

**Respuesta:** Que los métodos sobre la lista no dependen de condiciones de carrera. Luego si 2 procesos quieren agregar un nodo a esta lo harán de manera secuencial independientemente de cual de los dos llegue primero. No se modifica la lista por 2 o mas procesos al mismo tiempo.

- Si un programa utiliza esta lista atómica ¿queda protegido de incurrir en condiciones de carrera?

**Respuesta:** No necesariamente. Solo la lista está protegida de condiciones de carrera. El programa puede tener otras secciones desprotegidas de estos problemas.

- ¿Cómo hace su implementación de insertar para cumplir la propiedad de atomicidad?

**Respuesta:** Usamos un *while loop* donde el método `compare_exchange_weak` revisa si el siguiente nodo del nuevo nodo es igual a la cabeza actual de la lista atómica.

- Si lo es entonces reemplazamos la cabeza de la lista actual con el nuevo nodo y devolvemos *true*. Luego el *while* recibe *false* y termina el *loop*.
- Si no lo es entonces en "siguiete" de nuestro nodo actual guardamos la cabeza actual y devolvemos *false*. Luego el *while* recibe *true* e iteramos de nuevo para intentar hacer a nuestro nodo actual la cabeza.

## 2.2. Ejercicio 2

El objetivo del segundo ejercicio es implementar los métodos:

- `incrementar(string clave)`
- `claves()`
- `valor(string clave)`

### 2.2.1. Método: `incrementar(string clave)`

Este método incrementa en uno el valor almacenado para el elemento `clave`. Si la clave no existe, la agrega en la tabla hash y le asigna el valor uno.

Para ejecutarlo correctamente, el método debe realizar las siguientes acciones:

1. Aplicar la función hash a la clave para obtener el índice de la tabla hash.
2. Obtener la *listaAtomica* correspondiente al índice de la tabla hash de la clave.
3. Recorrer los elementos de la *listaAtomica* para verificar si el valor de algún *nodo*.*\_valor.first* coincide con la clave.
4. Si se verifica que existe algún nodo con esta clave, se incrementa *nodo*.*\_valor.second* en 1.
5. Caso contrario, con la función insertar vista en el ejercicio 1, se inserta la tupla `<clave, 1>` en la *listaAtomica*.

Ejecutados estos pasos, el método terminaría de forma correcta.

El enunciado requiere para este método que no incurra en condiciones de carrera. Por ejemplo, si dos procesos intentan incrementar el valor de la misma clave a la vez, podría suceder que los dos lean el valor actual simultáneamente. Entonces al incrementarlo se perdería el +1 de uno de los procesos. Para evitar esta condición se implementó un mutex, que previene que dos procesos modifiquen la tabla hash a la vez.

Una condición de sincronización pedida por el enunciado es que sólo haya contención en caso de colisión de hash. Esto imposibilita que se tenga un único mutex para toda la tabla hash, sino que se tenga un vector de mutex: un mutex por cada índice de la tabla. De esta forma se puede discriminar los índices de la tabla que quedan bloqueados de los que se pueden seguir accediendo, mejorando así la eficiencia del código.

Por último, se decide que ninguna función del ejercicio 2/3 pueda acceder a la clave si un proceso la está incrementando. Análogamente, ningún proceso puede incrementar la clave hasta que se verifique que no haya otro proceso accediendo a la misma. Para implementar esta condición, se decide convertir el vector de mutex explicado en el párrafo anterior en una matriz de mutex. Esta matriz, llamada *matrizMutexLectores*, es una matriz de 26 columnas y 3 filas. Cada columna representa un índice de la tabla hash y cada fila una de las funciones del ejercicio 2/3. En cada posición de la matriz se almacena un mutex.

	Tabla[0]		Tabla[i]		Tabla[25]
Claves()	Mutex[0][0]	...	Mutex[0][i]	...	Mutex[0][25]
Promedio()	Mutex[1][0]	...	Mutex[1][i]	...	Mutex[1][25]
Valor()	Mutex[2][0]	...	Mutex[2][i]	...	Mutex[2][25]

Figura 4: Esquema matrizMutexLectores

La función incrementar genera el *lock* de los tres mutex correspondientes a la columna de la clave. Esto lo realiza a partir de la función `lockAllReaders`. Así se logra la implementación de los requerimientos anteriormente descriptos.

### 2.2.2. Método: `claves()`

Este método devuelve todas las claves de una tabla hash.

Para ejecutarlo correctamente, el método debe realizar la siguiente acción:

1. Por cada índice de la tabla hash, se debe recorrer la *listaAtómica* correspondiente almacenando en la estructura de resultado el nombre de la clave. La misma coincide con *nodo.valor.first*.

Finalizado el recorrido, el método terminaría de forma correcta.

El método no implica la modificación de valores de la tabla hash. Por lo tanto, el único mecanismo de sincronización que se le aplica es hacer un *lock* al mutex de la fila `matrizMutexLectores[CLAVES]`. Esto se realiza con la función `lockTableFor`. De esta forma, nos aseguramos que iniciada la ejecución de la función `claves()`, ningún proceso va a modificar el valor de ninguna clave hasta que haya sido almacenada en el vector resultado.

Para evitar inanición, se decide que una vez almacenada la clave en el vector solución se genere el *unlock* de la clave. De esta forma los otros procesos no tienen que esperar a que termine la ejecución completa de esta función para poder ejecutarse.

### 2.2.3. Método: `valor(string clave)`

Este método devuelve el valor correspondiente al elemento `clave`.

Para ejecutarlo correctamente, el método debe realizar las siguientes acciones:

1. Se debe recorrer la *listaAtómica* correspondiente al índice de la tabla hash de la clave hasta encontrar el nodo que coincida con la clave. Esta se almacena en *nodo.valor.first*.
2. Se debe devolver el valor almacenado en el nodo. Este se almacena en *nodo.valor.second*.

Finalizados estos pasos, el método terminaría de forma correcta.

Este método tampoco requiere la modificación de ningún valor de la tabla hash. El único mecanismo de sincronización que se aplica es generar un *lock* en el mutex correspondiente a `matrizMutexLectores[CLAVES][indice]`, donde `indice` es el índice de la tabla hash correspondiente a la clave. De esta forma nos aseguramos que una vez iniciada la ejecución de la función `valor()`, ningún otro proceso modificará el valor de nuestra clave. Esto se realiza de una forma muy eficiente con la matriz, ya que el resto de las claves pueden ser accedidas.

#### 2.2.4. Preguntas del TP

Se procede a responder las preguntas indicadas en la consigna:

- ¿Cómo lograron que la implementación esté libre de condiciones de carrera?

**Respuesta:** Contamos con una matriz de mutex.

**Matriz de mutex (matrizMutexLectores):** Un arreglo de 26 mutex (uno por cada letra) para cada función de lectura. De esta manera si un proceso quiere escribir con *incrementar* en una entrada específica, estará obligado a esperar que todas las funciones de lectura liberen esa entrada. Solo se podrá modificar una lista enlazada, agregando un nodo nuevo o modificando uno actual, si ningún proceso de lectura está actualmente ocupando el recurso.

**Bloqueante?** Según la especificación se quiere que las funciones de lectura (valor, claves, promedio) sean no bloqueantes.

- *Valor()* solo bloquea una entrada de la tabla por lo que no es bloqueante de toda la tabla todo el tiempo de su ejecución.
- *Claves()* bloquea todas las entradas. Recolecta las claves correspondientes a la entrada “a”, desbloquea la entrada recorrida y pasa a la siguiente entrada para repetir el proceso. Bloqueamos toda la tabla para que ningún proceso pueda agregar alguna clave a mitad de camino de recolección y tengamos una inconsistencia entre las claves cuando llamamos a la función *Claves()* y las claves al retornar el arreglo. O sea que a medida que se juntan las claves de una entrada, estas se van desbloqueando evitando así un bloqueo de toda la tabla todo el tiempo de su ejecución.
- *Promedio()* Es igual que en *Claves()*. Bloqueamos toda la tabla, agarramos la información necesaria y desbloqueamos la entrada usada. Nuevamente es no bloqueante todo el tiempo de su ejecución.

Como se ve, *Claves()* y *Promedio()* bloquean toda la tabla solo al principio cumpliendo así lo pedido.

Decidimos usar una matriz en vez de un arreglo específico para cada lector para que a futuro sea extensible sin cambiar la implementación de las funciones. Simplemente se agrega la función de lectura deseada, se agrega a la lista de *ENUM* su nombre clave y se aumenta la cantidad de filas en 1 en la definición de la matriz.

**Pregunta del millón:** Por que se la complicaron tanto creando un arreglo de mutex para cada lector? Por que no todos los lectores comparten un arreglo de 26 mutex y listo? Supongamos que tenemos dos funciones lectoras y una función escritora que quieren operar sobre la entrada “a”.

- Solución un mutex para “a”: Para leer tengo que bloquear el mutex así la escritora no me modifica el dato pero a su vez estoy excluyendo a la otra lectora que no me lo va a modificar, no hace falta bloquear a otra lectora.
- Solución un mutex para lectoras de “a” y un mutex para escritoras de “a”: Para leer tengo que bloquear el mutex de escritura así la escritora no me modifica el dato. El tema es que la otra lectora también va a tener un *mutexEscritura.lock()* para bloquear a la escritora pero esta vez se va a “trabar” porque la primera de lectura ya bloqueo. Es como un fuego cruzado. Quedamos en la misma de antes.
- Solución final: Un mutex por lector. De esta manera las funciones lectoras bloquean **SU MUTEX** y no se cruzan entre sí. Ahora la función escritora solo puede escribir excluyendo a todas las lectoras y escritoras del recurso con sus respectivos mutex. Bloquea las escritoras porque si el proceso escritor 1 bloquea alguna de las lectoras entonces, al compartir código, el proceso escritor 2 queda bloqueado para escribir en esa misma entrada.

Supongamos que tenemos la tabla de la figura 5 y queremos devolver todas sus claves con la función *Claves()*. Se bloquean todas las entradas (representado con el color rojo) y se agregan de manera

progresiva desde la entrada “a” hasta la entrada “z”. Como dijimos en la definición, a medida que se agregan todas las claves de una entrada, esta se desbloquea (representado en verde) para que cualquier proceso que quiera escribir en esa entrada, pueda hacerlo pues no modifica el resultado de *Claves()*.

Vamos a seguir el ejemplo con un arreglo ficticio “claves”

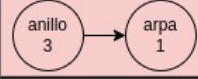

tabla[0]	a	
tabla[1]	b	
tabla[2]	c	ListaEnlazada2
...	...	...
tabla[i]	ASCII[i] + ASCII[a]	ListaEnlazada <i>i</i>
...	...	...
tabla[25]	z	ListaEnlazada25

Figura 5: Ejemplo de tabla con todas las entradas bloqueadas. claves = [].



tabla[0]	a	
tabla[1]	b	
tabla[2]	c	ListaEnlazada2
...	...	...
tabla[i]	ASCII[i] + ASCII[a]	ListaEnlazada <i>i</i>
...	...	...
tabla[25]	z	ListaEnlazada25

Figura 6: Todas las entradas bloqueadas menos la “a”. claves = [anillo, arpa].

tabla[0]	a	
tabla[1]	b	
tabla[2]	c	ListaEnlazada2
...	...	...
tabla[i]	ASCII[i] + ASCII[a]	ListaEnlazada <i>i</i>
...	...	...
tabla[25]	z	ListaEnlazada25

Figura 7: Todas las entradas bloqueadas menos la “a” y la “b”. claves = [anillo, arpa, balón].

- ¿Qué decisiones debieron tomar para evitar generar más contención o espera que las permitidas?

**Respuesta:** Con respecto a los métodos de *HashMapConcurrente*

- Bloquear de a una entrada para *incrementar/valor* agilizando las múltiples lecturas y escrituras sobre una misma tabla al mismo tiempo cumpliendo con los requisitos de la cátedra.
- Desbloquear las entradas ya leídas en *claves*. Por un lado evitamos condiciones de carrera pero por el otro liberamos recursos que ya no nos interesan, en este caso, claves ya leídas.

## 2.3. Ejercicio 3

El objetivo del tercer ejercicio es implementar los métodos:

- `promedio()`
- `promedioParalelo(unsigned int cantThreads)`

### 2.3.1. Método: `promedio()`

Este método devuelve el promedio de los valores de la tabla.

Para ejecutarlo correctamente, el método debe realizar las siguientes acciones:

1. Por cada índice de la tabla hash, se debe recorrer la *listaAtomica* correspondiente.
2. Por cada nodo se debe sumar el valor *nodo.\_valor.second* a la variable *suma*, e incrementar en uno el valor de la variable *count*.
3. Finalizado el recorrido por todas las *listasAtomica* de todos los índices, se debe devolver el resultado de dividir *suma* con *count*.

Finalizados estos pasos, el método terminaría de forma correcta.

El problema de sincronización para este método es similar a los vistos en el Ejercicio 2. Si se modifica el valor de algún nodo iniciada la ejecución de la función `promedio()`, el resultado entregado no será válido.

Para resolver este problema se recurre a la matriz `matrizMutexLectores` definida en el Ejercicio 2. El mecanismo de sincronización es similar al de la función `claves()`, ya que tampoco modifica la tabla hash, requiere el *lock* de toda la tabla al inicio de la función y a medida que recorre cada nodo le genera el *unlock*.

Se hace entonces un *lock* al mutex de la fila `matrizMutexLectores[PROMEDIO]`. Esto se realiza con la función `lockTableFor`. De esta forma, nos aseguramos que iniciada la ejecución de la función `promedio()`, ningún proceso va a modificar el valor de ninguna clave hasta que haya sido incluida en las variables *suma* y *count*.

Para evitar inanición, se decide que una vez almacenada la clave en el vector solución se genere el *unlock* de la clave. De esta forma los otros procesos no tienen que esperar a que termine la ejecución completa de esta función para poder ejecutarse.

### 2.3.2. Método: `promedioParalelo(unsigned int cantThreads)`

El método `promedioParalelo` como indica su nombre, procesará la tabla con idealmente dos o más threads corriendo en simultáneo con el objetivo de poder calcular el promedio de una forma más rápida y eficiente. Hay que tener cuidado de contar todas las letras sin repeticiones y para ellos se sincronizarán los diferentes hilos con una variable atómica. Al igual que `promedio`, se hace el mismo bloquea en la matriz, pero esta vez los hilos en la función auxiliar `contadoresThread` se encargan de desbloquearla a medida que terminan una letra.

### 2.3.3. Preguntas del TP

Se procede a responder las preguntas indicadas en la consigna:

- ¿Qué problemas puede ocasionar que promedio e incrementar se ejecuten concurrentemente?

**Respuesta:** Puede ocurrir que se inserte un nodo nuevo o se incremente un valor en una posición que todavía no consideramos para los cálculos del promedio actual generando así un nuevo promedio diferente al promedio real que tenía el HashMap al momento de llamar a la función.

- Piensen un escenario de ejecución en que promedio devuelva un resultado que no fue, en ningún momento, el promedio de los valores de la tabla.

**Respuesta:** Supongamos que llamamos a la función promedio, y mientras se esta recorriendo la tabla otro thread inserta un valor que ya existe en una entrada que aún no fue barrida, aumentando en 1 su cantidad de pedidos. La función retornará un promedio diferente al promedio real al momento de llamar a la sub-rutina.

- Implementación de promedio paralelo

**Respuesta:** Se crea una variable atómica para compartir entre los diferentes threads que irán incrementando atómicamente para recorrer las entradas de la tabla. A su vez cada thread contará con contadores de total de palabras y entradas únicas que luego serán sumadas en su totalidad para poder calcular el promedio real. Porque el promedio de los promedios **NO ES** el promedio real.

- ¿Tiene la misma firma (signature) que la función original promedio o devuelve tipos distintos?

**Respuesta:** Sí, tiene la misma firma.

- ¿Cuál fue la estrategia elegida para repartir el trabajo entre los threads?

**Respuesta:** Que cada uno sume de manera propia sus valores y cantidad de nodos. Cada thread se encarga de ninguna (dependiendo de los hilos creados y la repartición que haga el scheduler sobre el trabajo con los hilos) o al menos una entrada de letra diferente.

- ¿Qué recursos son compartidos por los threads?

**Respuesta:** Variable atómica *pos* y *matrizMutex* pero a diferentes entradas del arreglo de mutex correspondiente a la función de lectura *promedio*.

- ¿Cómo hicieron para proteger estos recursos de condiciones de carrera?

**Respuesta:** Variable atómica para sincronizar la asignación de índices en las entradas de la tabla con la ayuda de *matrizMutex* correspondiente al arreglo *promedio* sincronizando así con los lectores y escritores.

## 2.4. Ejercicio 4

- ¿Fue necesario tomar algún recaudo especial, desde el punto de vista de la sincronización, al implementar cargarCompras? ¿Por qué?

**Respuesta:** Desde el punto de vista de *CargarCompras* nos abstraemos de la sincronización pues es trabajo del *HashMapConcurrente* el incrementar las palabras. Luego recorrer un único archivo es lineal y no hay problemas de concurrencia. Es línea a línea. Se tuvo que modificar la implementación dada por la cátedra porque esta se hacía palabra por palabra. Por ejemplo “Anteojo 3D” se interpretaba como “Anteojo” y “3D” cuando la idea era leer “Anteojo 3D”. Nuestra implementación guarda minúsculas y mayúsculas por separado.

- Decisiones tomadas para la implementación de cargarMultiplesArchivos

**Respuesta:** Es bastante parecido a *promedioParalelo* donde se declara una variable atómica para ir indexando las múltiples compras a procesar. Cada thread usa la función cargar compra del archivo que se le vaya asignando.

### 3. Análisis del sistema

- ¿Si su sistema tuviera que ser usado bajo durante el *Black Friday* con millones de compras a la vez, estaría satisfecho con la solución actual?

**Respuesta:** Para ver el comportamiento de nuestra implementación se procedió a crear un ejecutable nombrado *Intensivo* con el cuál poder observar la *performance* de nuestro trabajo. Creemos que nuestra implementación es óptima en cuanto a inserciones, sobre todo cuando se trabaja con múltiples threads y lectura de valores de las claves. Mientras tanto, podemos observar que al pedir *promedioParalelo* o *claves* por su naturaleza, donde para garantizar un resultado válido empiezan bloqueando la tabla y liberan a medida que procesan, y que además recorren la tabla en su completitud, vemos como el rendimiento decrece significativamente, sobre todo con claves al ser una función que utiliza un solo thread. A su vez esta reducción de rendimiento afecta las inserciones severamente. Teniendo en cuenta que en el *Black Friday* se harán muchos pedidos (o sea, inserciones), que no es necesario requerir constantemente claves o promedio, ya que son herramientas más útiles para un análisis de datos después del evento, y que valor se correrá a manera de control regularmente espaciado en el tiempo, pero no afecta tanto al rendimiento de la inserción, consideramos nuestro software podría sobrevivir a este día.

- ¿Se le ocurre algún inconveniente o oportunidad de mejora?

**Respuesta:** Como mencionamos anteriormente, al bloquearse la tabla completamente para la escritura tanto al pedir claves como pidiendo promedio, de ejecutarse constantemente estas funciones, el flujo de entrada de inserciones se vería afectado gravemente. Al requerirse tener un valor concreto al momento que estas funciones son ejecutadas, este será un problema constante que se tendrá mientras el requerimiento no cambie. No obstante, creemos que las inserciones se podrían acelerar mínimamente eliminando complejidad en los mutex, cambiando la matriz que permite distintas operaciones de lectura (promedio, claves, valor) simultáneamente o una inserción, por un solo arreglo de mutex donde solamente se se pueda realizar una operación. Esto, en un contexto como el *Black Friday* donde casi exclusivamente estaríamos realizando inserciones, nos ahorraría un par de instrucciones de CPU ya que actualmente debemos tomar tantos mutex como funciones que leen hay (3), que multiplicado por la cantidad de compras constantes podría acelerar las inserciones cuando se habla en el orden de millones de operaciones. Como desventaja solo se podría realizar una operación a la vez, ya que todas las funciones serían exclusivas entre ellas. Otro punto que encontramos se podría mejorar, pero en detrimento de la inserción y por lo tanto no sería implementado para un *Black Friday*, es que si bien se pueden realizar hasta tres operaciones de lectura diferentes, no podemos realizar dos o más del mismo tipo, o sea no podemos hacer varios promedios, claves ni valor en simultaneo. Esta última operación siendo la que más nos gustaría poder ejecutar dos o más instancias en simultáneo que deban entrar en la misma entrada de letra de la tabla. Una posible solución sería contar con una variable atómica controlada por insertar, que permita o no a una instancia de valor entrar a una entrada de la tabla, y otra variable atómica a modo de contador controlado por valor que indique cuantas operaciones se están realizando en una determinada entrada y mientras hayan operaciones del tipo valor, no se realizará una inserción en dicha entrada de tabla. Debido a la penalización que generan claves y promedio, esto no debería ser implementado para dichas funciones.

- En el caso de que la tienda tenga solo dos tipos de productos, ¿Recomendaría el uso de concurrencia? ¿Por que?

**Respuesta:** En el contexto de funciones paralelas no lo recomendaríamos porque la creación de threads puede ser mas costosa que usar las funciones normales donde el mutex protege la zona critica. Depende mucho del *overhead* que genera la creación de threads. Al ser dos productos asumimos que los métodos secuenciales son la mejor solución.

- Implementación de *calcularMedianaParalela*

**Respuesta:** Considerando que la mediana es el elemento que se encuentra en el medio (puede variar para pares e impares) de una lista ordenada. Lo que queremos hacer es juntar en un arreglo todas las cantidades de productos de forma que estén ordenadas y de eso obtener la cantidad del medio. Consideramos que las únicas cantidades que pueden estar en el arreglo son las que ya se encontraban al momento de llamar al método. O sea, que no se puede incrementar una cantidad al momento de ejecución de esta rutina. Además se nos pide que sea de manera concurrente. La solución que proponemos es copiar la implementación de *promedioParalelo()* donde se bloquea toda la tabla y se va desbloqueando de la “a” a la “z” a medida que cada thread termina de recolectar en un arreglo las cantidades de una entrada. Al final todo se junta en un único arreglo. Este se ordena y se busca el valor del medio para calcular la moda.

- Analizar en detalle qué dificultades podrían enfrentar al implementar.

**Respuesta:** El único problema de la implementación es que es costoso por el tema de crear un arreglo para cada entrada e ir cargándolo. Además tengamos en cuenta que una entrada puede tener millones de elementos, habría que tener cuidado con el tamaño del *stack* porque se podría generar un *stack overflow*. Otro problema es el tiempo. Como dijimos cada entrada puede tener muchos elementos en su lista atómica así que recolectar las cantidades puede generar un bloqueo largo. Por el resto no hay mucho cambio con respecto a *promedioParalelo()*, los cambios a la implementación son simples.

Hay que crear un arreglo de *int* para cada entrada y pasarlo como parámetro por referencia. Cuando un thread entra a lo que sería el actual “*contadoresThread*”, agrega la cantidad que se va recorriendo y sigue hasta que no haya más entradas.

Luego podemos crear una variable “*cant.elementos*” para acumular en ella la cantidad de elementos que suman en total cada arreglo de cada entrada.

Podemos usar un *while* de  $i = 0 \dots \text{cant.elementos}$  donde definimos un *for* para cada arreglo, en estos *for* vamos agregando en un arreglo general los elementos del arreglo de la entrada “a”, después los elementos del arreglo de la entrada “b” y así sucesivamente hasta la “z”. (Es una forma de concatenar todos los arreglos en uno solo)

Luego usamos la función de la *stdlib* para ordenar el arreglo y elegimos la mediana. Hay que tener en *consideración* que si el arreglo es par entonces elegimos la mediana aritmética (los dos del medio dividido 2) y si es impar simplemente el del medio. El resultado es un *float* porque la media *aritmética* puede no ser entero.

- ¿Qué estrategias adoptarías para minimizar el tiempo de espera de los threads, la comunicación entre procesos y evitar condiciones de carrera?

**Respuesta:** Lo mismo que *promedioParalelo()*. Bloqueamos toda la tabla para escritores desde la “a” a la “z” y vamos desbloqueando a medida que se va recorriendo. De esta manera múltiples threads pueden entrar en la tabla y acelerar el proceso de barrida. Es una estrategia que asegura la integridad de los datos.



Figura 8: src: <https://quii.gitbook.io/learn-go-with-tests>